PARALLEL ALGORITHMS FOR SOME PROBLEMS ON GRAPHS

by

SUDARSHAN BANERJEE

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

FEBRUARY, 1995

Parallel algorithms for some problems on graphs

A Thesis Submitted

in Partial Fulfilment of the Requirements

for the Degree of

MASTER OF TECHNOLOGY

by

Sudarshan Banerjee

to the

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

February, 1995

CSE-1995-M-KAN-[illegible]

Certificate

This is to certify that the work contained in the thesis titled **Parallel algorithms for some problems on graphs** by **Sudarshan Banerjee (Roll No: 9311127)**, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Feb 6, 1995

[illegible]

Dr. Sanjeev Saxena
Assistant Professor,
Department of Computer
Science and Engineering
IIT, Kanpur

Acknowledgements

I wish to take this opportunity to express my sincere gratitude to my thesis advisor, Dr. Sanjeev Saxena for providing me guidance in all aspects of thesis work and for being patient during my idle, unproductive phases. Also, I wish to thank my examiners, Dr. R.K.Ghosh and Dr. S. Pushpavanam for their constructive suggestions which improved the presentation of this thesis.

Next, I want to thank my mother who was the instrumental factor in my joining the M.Tech programme and thus being involved in academics again, albeit for a short period.

Last, but definitely not the least, I would like to thank my friends (M.Tech '93 batch of CSE, and Hall-5 residents) and the environment (sports and academics) at I.I.T Kanpur for making my stay here the most enjoyable phase of my life to date.

[illegible]

(SUDARSHAN BANERJEE)

Abstract

In this thesis, parallel algorithms have been obtained for some problems on graphs.

1. It is shown that the problems of computing Maximum Independent Set, Minimum Dominating Set and Minimum Clique Cover on Interval Graphs can be solved in $O(1)$ time on a $n \times n$ reconfigurable mesh.
2. A parallel algorithm requiring $O(\log m)$ time using m processors on the CREW PRAM is obtained for the *most vital edge* (MVE) problem. $G = (V, E)$ is a weighted undirected graph with n vertices and m edges, each edge e having a weight $w(e)$ assigned to it. If $f(G)$ is the weight of a minimum spanning tree of G the MVE of G is an edge e such that $f(G-e) \geq f(G-e')$ for every other edge e' of G. Sequential algorithm for the problem takes $O(\min(m+n\log n, m\alpha(m,n))$ time[18].
3. A parallel solution is obtained for computing a DFS tree in a permutation graph. The algorithm takes $O(\log^3 n)$ time using n processors on a CREW PRAM or $n\frac{(\log\log n)^2}{\log n}$ processors on an ARBITRARY-WRITE PRAM.
4. A parallel solution is obtained for computing a BFS tree in a permutation graph. The algorithm takes $O(\log n)$ optimal time using $\frac{n}{\log n}$ processors on an EREW PRAM.
5. Optimal parallel algorithms are described for computing the connected and biconnected components of a permutation graph. The algorithms can be implemented in $O(\log\log\log n)$ time with $\frac{n}{\log\log\log n}$ processors on a COMMON (or TOLERANT) CRCW PRAM, or alternatively in $O(\log n)$ time with $\frac{n}{\log n}$ processors on a CREW PRAM. Algorithm for connected components can also be implemented in $O(\log^* n)$ time with $\frac{n}{\log^* n}$ processors on a Priority-Write PRAM.

Contents

Chapter 1

Introduction

A reference to an *algorithm* normally implies a sequential algorithm, i.e a single instruction can be executed in an instant of time. In recent years, there has been an increasing awareness of the bottleneck in sequential computing, namely physical laws (minimum size, minimum distance between transistors) which determine the maximum speed a sequential computer (based on transistors as the basic computing element) can achieve.

To overcome this bottleneck, i.e obtain faster running times, the concept of *parallel computing* has been introduced as a viable alternative. The central idea is to execute multiple steps of the given algorithm simultaneously; this expectedly reduces the total computation time. A parallel computer does this by using multiple processing units (processors). A problem to be solved is partitioned into subproblems which are then assigned to different processors so that they can be executed simultaneously; once these computations are over, the solutions are combined to obtain the solution to the original problem. A *parallel algorithm* is a solution method for solving problems on a parallel computer [3].

In this thesis, parallel algorithms are presented to solve some problems belonging to an important class, namely graphs. Graphs were first introduced by Euler in 1736 and, since then have found innumerable applications in almost all domains of Engineering and Science [12]. In the next setion, we take a look at some graph-theoretic preliminaries where we describe some common terms with which the reader will be assumed to be familiar throughout the rest of this thesis.

1.1 Graph-theoretic preliminaries

The definitions here are mostly consistent with [14].

- **Graph** A graph $G = (V, E)$ consists of a finite set of vertices V and a finite set of edges E where each edge is an ordered pair, (x, y), $x, y \in V$. The graph G is said to be *undirected* if the edge relationship is symmetric, i.e $(x, y) \in E \Leftrightarrow (y, x) \in E$. Subsequently, the term graph refers to undirected graphs only. A *subgraph* of $G = (V, E)$ is any graph $G' = (V', E')$ where $V' \subseteq V$ and $E' \subseteq E \bigcap (V' \times V')$.
- **Isomorphism** Two graphs $G = (V, E)$ and $G' = (V', E')$ are called isomorphic, if there is a bijection $f : V \rightarrow V'$ such that $\forall x, y \in V$, $(x, y) \in E \Leftrightarrow (f(x), f(y)) \in E'$.
- **Complement** The complement of the graph $G = (V, E)$ is the graph $G' = (V, E')$ where $E' = \{(x, y) \in V \times V \text{ and } (x, y) \notin E\}$.
- **Path** A path between two vertices v_1 and v_l in G is a sequence of vertices $[v_1, v_2, ..., v_l]$ such that $(v_i, v_{i+1}) \in E \;\; \forall i \in [1..(l-1)]$.
- **Cycle** A cycle is a sequence of vertices $[v_1, v_2, ..., v_l, v_1]$ such that $(v_i, v_{i+1}) \in E \;\; \forall i \in [1..(l-1)]$ and $(v_l, v_1) \in E$.
- **Connected graph** A graph is connected if there is a path between any two vertices in the graph.
- **Tree** A tree $T = (V, E)$ is a connected acyclic graph.
- **Minimum spanning tree** Let $G = (V, E)$ be an undirected connected graph where each edge has a weight assigned to it. A subgraph $G' = (V, E')$ is a minimum spanning tree iff G' is a tree and the sum of the weights of edges belonging to G' is minimum among all such trees.
- **Clique** A clique in G is a subset V' of the vertex set V of G such that there is an edge between every pair of vertices in V', i.e $x, y \in V' \Rightarrow (x, y) \in E$. A clique is *maximal* if no other clique of G properly contains it. A clique is

maximum if the cardinality (number of vertices) of the clique is not less than the cardinality of any other clique of G; this cardinality is denoted as $\omega(G)$. A *minimum clique cover* is a partition of the vertices in V such that each subset is a clique; additionally, the number of subsets is minimum– this number is denoted as $k(G)$.

- **Independent set** An independent set in G is a subset V' of the vertex set V of G such that there is no edge between any pair of vertices in V', i.e $x, y \in V' \Rightarrow (x, y) \notin E$. The number of vertices in an independent set of maximum cardinality is denoted as $\alpha(G)$. A *colouring* is a partition of the vertices in V such that each subset is an independent set. The *chromatic number* of G is a colouring in which the number of such subsets is minimum– this number is denoted as $\chi(G)$.
- **Perfect graph** In any graph, $\omega(G) \leq \chi(G)$. The class of undirected graphs for which $\omega(G) = \chi(G)$ is termed as the class of perfect graphs. This is a well-studied class of graphs [6] and important from a practical view-point. Note that many graph problems which are NP-complete for general graphs, e.g, finding $k(G)$, are solvable in polynomial time for graphs belonging to subclasses of this class; even linear time algorithms are possible, e.g for interval graphs.

In the next two subsections, we consider two important subclasses of perfect graphs for which parallel algorithms are designed in this thesis.

1.1.1 Permutation graphs

We consider a permutation π of the n integers in the range $[1..n]$. For an integer i, $\pi^{-1}(i)$ denotes the position in π where i occurs. The graph $G(\pi) = (V, E)$ is defined in the following manner: for each integer $i \in [1..n]$ there is a corresponding vertex in V and an edge $(i, j) \in E$ iff $(i - j)(\pi^{-1}(i) - \pi^{-1}(j)) < 0$. An undirected graph G is called a permutation graph iff there exists $G(\pi)$ for some π, such that G is isomorphic to $G(\pi)$.

A common representation of permutation graphs is the *matching diagram* [14]. This consists of two rows of integers; the upper row, U is the sequence of all integers

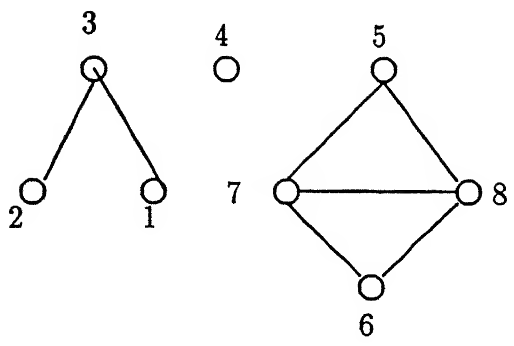

Fig 1a. Permutation graph

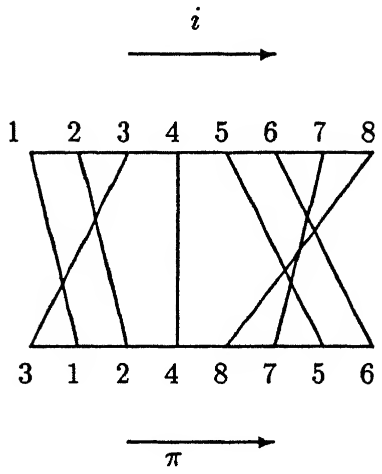

Fig 1b: Matching diagram

in the range $[1..n]$ while the lower row D is the given permutation, π. Each integer i in U is connected by a straight line to that position j in D such that $\pi(j) = i$; thus we may view the graph as a bijective mapping of integers from U to D; an edge (i, j) is in the graph if and only if the lines i and j intersect in the matching diagram. In Fig 1 we have an example showing the permutation graph and matching diagram corresponding to $\pi = (3, 1, 2, 4, 8, 7, 5, 6)$.

This class of graphs has been well-studied [13] and is an important subclass of perfect graphs with a large number of practical applications. Before proceeding to give an example, we recall some of the important properties of graphs belonging to this class: (a)they are transitively orientable, i.e each edge can be assigned a direction such that the resulting oriented graph (V, F) satisfies the following property: $(x,y),(y,z) \in F \Rightarrow (x,z) \in F$, where $x,y,z \in V$ (b) their complement also belongs to this class.

Application Consider a set of cities which can be partitioned into two subsets S_1 and S_2 such that all cities in S_1 (also S_2) lie along a straight line. There are flights between cities in S_1 and S_2 scheduled at the same time. The problem is to assign altitudes to each flight path so that intersecting paths are at different altitudes.

To solve this problem, a bipartite graph is constructed with S_1 and S_2 as the two parts; corresponding to each city there is a vertex, and there is an edge between a vertex in S_1 and a vertex in S_2 if there is a scheduled flight between the two cities corresponding to the vertices. From this bipartite graph, a matching diagram [14] can be easily extracted (by splitting each vertex into multiple vertices, one for each edge incident on it), i.e the given problem can be framed as a permutation graph problem. The modified problem is to compute a colouring of the given graph, so that vertices which have an edge connecting them (or flight paths intersect) are assigned different colours (altitudes).

1.1.2 Interval graphs

Consider a set of intervals on the real line (or in general, any linearly ordered set). Corresponding to each interval, there is a vertex and two such vertices have an edge between them iff their corresponding intervals intersect. Such a graph is called an interval graph. To put it slightly more formally, if the interval I_i is represented as $I_i = [a_i, b_i]$, $(a_i < b_i)$, there is an edge in the interval graph between vertices I_i and I_j iff either $a_j < a_i < b_j$ or $a_j < b_i < b_j$, where we make the simplifying assumption (without any loss of generality) that each a_i, b_i, a_j, b_j is distinct.

This is an important subclass of perfect graphs with many applications in practical domains. As an example we consider the following application [14].

Application There are n medicines, M_1,M_2,..M_n which need to be stored at suitable temperature levels; to be more exact, the temperature for storing medicine M_i needs to be kept constant and must lie between T_i and T_i'. The problem is to find out the minimum number of refrigerators needed to store all medicines.

An interval graph G is constructed in the following manner: for each vertex M_i, $i \in [1..n]$ there is a corresponding interval $I_i = [T_i, T_i']$. Now, if we consider some temperature T, all intervals which pass through T have edges with each other(i.e, they form a clique) and the corresponding medicines can be kept in the same refrigerator. Thus, to solve the given problem, we need to find out the minimum number of cliques partitioning the vertex set or, the minimum clique cover has to be computed in G.

1.2 Computation models

The models of parallel computation on which the algorithms in this thesis have been designed are the Parallel Random Access Machine (*PRAM*) model and the *reconfigurable mesh* model.

A reconfigurable mesh of size $m \times n$ consists of an $m \times n$ array of processors connected by a reconfigurable bus system. Each processor has four ports besides its arithmetic and logic unit. By adjusting the local connections between the ports of a processor, the bus system can be dynamically configured. Communication is carried out between processors by writing or broadcasting values on the bus. This model is more thoroughly defined in Chapter 2.

In the PRAM model, there are a finite number of processors (RAM's) operating synchronously on an infinite global(shared) memory. It is assumed that each processor has an unique index which it knows. All processors execute the same program; in each step, a processor may carry out local computation, or it may access the global memory.

Depending on the nature of access to the memory, this model can be further divided into subclasses. The specific subclasses referred to in this thesis are:

Exclusive Read, Exclusive Write (*EREW*): No two processors can access the same memory location in a single unit of time.

Concurrent Read, Exclusive Write (*CREW*): Multiple processors can access the same memory location simultaneously for reading, but exclusivity is maintained while writing.

Concurrent Read, Concurrent Write (*CRCW*): Multiple processors can read from as well as write into the same memory location simultaneously. If concurrent write is allowed, there needs to be some conflict-resolution rules to determine the unique value contained in a memory location where multiple processors have tried to write simultaneously. The further subclasses of the CRCW model referred to in this thesis are:

ARBITRARY: As the name denotes, some arbitrary processor succeeds when multiple processors are trying to simultaneously write different values to the same location.

TOLERANT: If multiple processors try to simultaneously write to the same location, the contents of the location do not change.

COMMON: All processors trying to simultaneously write to the same location must attempt to write the same value which then gets stored.

PRIORITY: Among the processors trying to simultaneously write to the same location, the least numbered processor succeeds.

1.3 Overview of thesis

In Chapter 2, constant time solutions are obtained for some problems in interval graphs on the reconfigurable mesh model of computation. In Chapter 3, the problem of computing the *most vital edge* of a graph is considered and an $O(\log m)$ time, m processor solution is described on the CREW PRAM. In Chapter 4, the problems of parallel construction of a DFS tree and a BFS tree for a permutation graph are considered. The construction of a BFS tree takes $O(\log n)$ optimal time using $\frac{n}{\log n}$ processors on an EREW PRAM while the construction of a DFS tree takes $O(\log^3 n)$ time with n processors on a CREW PRAM. In Chapter 5, fast optimal solutions are described for computing the connected and biconnected components of a permutation graph. These solutions take $O(\log\log\log n)$ time with $\frac{n}{\log\log\log n}$ processors on a COMMON (or TOLERANT) CRCW PRAM. Some conclusions are finally offered in Chapter 6.

Chapter 2

Constant Time Algorithms for Interval graphs on Reconfigurable mesh

2.1 Introduction

Interval graphs model a large number of real-life problems in scheduling and VLSI; for e.g, the intervals can be job-starting times in a job-scheduling problem or positions of components in a VLSI routing problem. It is therefore important to design parallel algorithms to obtain fast solutions for problems on these graphs. In this chapter, we look at the techniques needed to solve these problems on processor arrays with reconfigurable bus systems or *reconfigurable mesh*. It is possible to develop constant time, i.e $O(1)$ algorithms, on this model while there are very few non-trivial problems which have a constant time solution on the CRCW, the most powerful PRAM model.

The reconfigurable mesh is a powerful model of computation– the 2-d mesh has been shown to be at least as powerful as the CRCW model of parallel computation [33]. In fact, $O(1)$ time algorithms are known for some problems (on reconfigurable mesh) for which a lower bound of $\Omega(\frac{\log n}{\log\log n})$ on time is known on CRCW model (with polynomial processors); these include the problems of computing transitive closure of an undirected graph which can be done in $O(1)$ time on a 3-d mesh with

$n \times n \times n$ processors, or on a 2-d mesh with $n^2 \times n^2$ processors [32]; n numbers can be sorted in $O(1)$ time on a 2-d mesh using $n \times n$ processors[19]; these results imply $O(1)$ time algorithms for various graph problems like bipartite graph recognition and computing biconnected components within the same resource bounds. An important aspect of this model is the fact that a VLSI chip, YUPPIE [21] has been developed to demonstrate the concepts of the polymorphic-torus network which is in functionally the same class as the model we use. The reconfiguration is achieved at a fine-grained level by circuit-switching– ports connected together are literally 'shorted'.

In this chapter we consider the problem of obtaining $O(1)$ time implementations of some interval graph algorithms on the 2-d mesh. Specifically, we consider the maximum independent set, minimum clique cover and minimum dominating set problems; the algorithms for each of these problems take $O(\log n)$ time with n processors on an EREW PRAM. As in general, intervals need not be sorted, the set of intervals provided need to be sorted, which leads to these resource bounds. We obtain implementations using $O(n^2)$ processors and constant time, which matches the resource bounds for sorting.

This chapter is organized as follows: in Section 2.2, the reconfigurable mesh model is described. In Section 2.3 we present solutions to some problems on the reconfigurable mesh. In Section 2.4 we show how these solutions can be applied to solve the interval graph problems. Finally, in Section 2.5 we conclude with some final remarks.

2.2 Reconfigurable array in 2-d

In this chapter, we use the "standard" reconfigurable mesh proposed by Miller et.al. [25]. A mesh of size $m \times n$ consists of $m \times n$ array of processors connected by a reconfigurable bus system. Each processor has the capability of performing basic arithmetic and logical operations and it is assumed that each such basic operation can be performed in one time unit, i.e., a comparison or an addition involving two numbers would take one unit of time. The processors are identified by their positions on the grid, i.e. $PE(i,j)$, $1 \leq i \leq m, 1 \leq j \leq n$, refers to the processor situated at the intersection of the ith row and the jth column; each processor is aware of its identity.

Each processor has four ports denoted as U, D, L, R (i.e., up, down, left, right respectively) besides its arithmetic and logic unit. These ports are the actual connection between the processor and the reconfigurable bus system; by adjusting the local connections between the ports of a processor, the bus system configuration can be changed dynamically (or reconfigured); for example, by connecting the U and D ports of each processor internally we obtain vertical buses; if we now disconnect some of the connections, we can split a vertical bus into subbuses.

Communication is carried out between processors by writing or broadcasting values on the bus (each bus is assumed to be capable of carrying $O(\log n)$ bits of data, where n refers to the size of the problem). Each broadcast is assumed to take one unit of time; further in a single unit of time, only one processor is allowed to write or broadcast on a subbus shared by multiple processors to avoid write conflicts. In case more than one processor attempts to broadcast, the final value on the bus is indeterminate.

2.3 Basic Routines for Reconfigurable Mesh

Some of the important routines we will be using in this chapter are

- **Minimum** Minimum of n numbers can be computed in $O(1)$ time on a $n \times n$ mesh [24].
- **Sorting** On a $n \times n$ mesh, n elements can be sorted in $O(1)$ time [19].
- **Nearest 1.** Given a sequence of 0's and 1's, for each 0 it is required to find the nearest 1 to its left. The problem can be solved on a one-dimensional reconfigurable mesh (first row of 2-dimensional mesh) using bus-splitting technique [24]. We construct a horizontal bus and then split this bus to the left of each 1. Now if each processor which contains a 1 broadcasts its identity on the bus to its right, the problem is solved.

- **Prefix-minima(maxima).** The problem of computing Prefix-minima(maxima) of n numbers, is considered in Subsection 2.3.2.
- **Mark all ancestors.** Given a (rooted) forest, and a distinguished node v, the problem is to mark all ancestors of v, i.e., to mark all nodes from v to the root of the tree containing v. Solution to this problem is described in Subsection 2.3.1.

2.3.1 Marking Ancestors

We are given n nodes each of which contains a variable *link* referring to the identity of its parent in the forest (successor in linked list). We are also given a distinguished node v, and it is required to mark all ancestors of v in the forest, (its successors in linked list), i.e. we have to mark all elements lying between v, the chosen element and the root of the tree to which v belongs. This is a modification of the $O(1)$ time algorithm for connected component problem which requires a $n \times n \times n$ or $n^2 \times n^2$ mesh[32]; we will be using an $n \times n$ mesh. We assume that input is in the topmost row, i.e. the processors $PE(1,i)$ contain the input (the *link* variables).

By connecting L and R ports of each processor, we will get a horizontal bus in each row; if j is an ancestor (successor) of element i, we connect rows i and j by means of a vertical bus in column j. Therefore, if $link(i) = j$ and $link(j) = k$, we see that a data path exists between elements i and k. Proceeding inductively, it can be easily shown that a data path exists between an element and all its ancestors in the tree; further the data buses of different trees are disjoint. To ensure that proper descendants of v are disconnected, we disconnect all vertical U and D links of row containing v, except from v to its parent.

Less informally the algorithm is:

Step 1 Let us assume that the input is in the 1'st row. Each processor $PE(i,i)$, $1 \leq i \leq n$ sets up the local connection $\{U,L\}$. In each row i, processors $PE(i,j)$, $1 \leq j < i$, set up the connection $\{L,R\}$ and processors $PE(i,j)$, $i < j \leq n$, set up the connection $\{U,D\}$. The net effect is to set up L-shaped buses.

Step 2 Each processor $PE(1,i)$ transmits its link value to processor $PE(i,1)$ using the bus connection set up in Step 1.

Step 3 The buses are disconnected and straight horizontal buses are formed by connecting the $\{L,R\}$ ports of each processor.

Step 4 Each processor $PE(i,1)$ transmits the value it received in Step 1 to all the processors in its row. Thus, for an element initially present in the i'th processor(of the first row) all n processors of the i'th row know its successor after execution of this step.

Step 5 Each processor sets up the connection $\{U,D\}$; now there are n horizontal and n vertical buses.

Step 6 Each processor $PE(i,j)$ where j is the successor of i in the linked list sets up the local connection $\{U,L\}$, thus connecting the i'th horizontal bus to the j'th vertical bus. Also, each processor $PE(i,i)$ sets up the local connection $\{U,L\}$ to make a path between horizontal buses i and j.

In this step, the elements in a linked list are connected in such a manner that if any one of them broadcasts some value, all elements in the list receive it. We are in effect computing transitive closure: if there is a data path $i \rightarrow j$ and link(j) $= k$, then there is also data path $i \rightarrow k$.

Step 7 Processor $PE(h,h)$ where h is the start point in the linked list, disconnects its $\{U,L\}$ connection. Then it transmits any value. All processors which receive this value are our desired successor elements.

2.3.2 Prefix Minima

We will use an $n \times n$ mesh to carry out the computation. Initially, n elements are in n processors of the topmost row. For each i, we will try to identify elements whose value is less than the value of element i and which also occur before i in the list; the minimum of these elements is the prefix-minima for element i. We can sort the items to get rank of all elements (rank of i is the number of elements less than i); we will assume that sorted items are again in topmost row. We next send the elements in topmost row down the column; as a result, contents of each row are identical to that of the topmost row. Items to the left of item i are less than item i. If we mark items, now left of i, which were originally left of i, and find the first (leftmost) marked item, we are done.

In more detail,

Step 1 We sort given elements on key (*val*,*pos*) where *val* is the value of the element and *pos* is the initial position of the element in the unsorted list [19], such that processors in first row contain elements in sorted order.

Step 2 Each processor $PE(i,i)$, $1 \leq i \leq n$ sets up the local connection $\{U,L\}$. In each row i, processors $PE(i,j)$, $1 \leq j < i$, set up the connection $\{L,R\}$ and processors $PE(i,j)$, $i < j \leq n$, set up the connection $\{U,D\}$. The net effect is to set up L-shaped buses.

Step 3 Each processor $PE(1,i)$ transmits *pos* to $PE(i,1)$ using the bus connection set up as above.

Step 4 Each processor disconnects its previous connection and sets up the connection $\{L,R\}$ forming horizontal buses.

Step 5 Each processor $PE(i,1)$ transmits *pos* to all the processors $PE(i,j)$ in its row. At the end of this step, all the processors in the i'th row know the initial position of the element whose sorted rank is i.

Step 6 Each processor disconnects its $\{L,R\}$ connection and sets up its $\{U,D\}$ connection forming vertical buses.

Step 7 Each processor $PE(i,i)$ transmits *pos* to all the processors in its column; let this new value of *pos* received by the processors be denoted as *pos'*.

Step 8 Each processor $PE(i,j)$ compares *pos* with *pos'*. If $pos < pos'$ and $j < i$, it sets a boolean flag to 1, else to 0. Effectively, in column i, the elements which have a sorted rank less than i and their unsorted position is also less than i have their flags set to 1; the remaining elements have their flags set to 0.

Step 9 In each column we find out the first occurrence of a 1. For this, we use bus-splitting in a similar fashion as in the algorithm of Nearest 1. The vertical buses are split at each 1 and the processors with 1's transmit their identity to the upper subbus through their U port. The value received in $PE(1,i)$ is the prefix minima corresponding to element i.

2.4 Interval graphs

2.4.1 Definitions

Given a family $\mathcal{I} = \{I_i, I_2, ...I_n\}$ of intervals, after Olariu [27], we define $first(\mathcal{I})$, $right(I_i)$ and $next(I_i)$.

1. The parameter $first(\mathcal{I})$ is the interval in family $\mathcal{I}$ which ends first :
   $first(\mathcal{I}) = I_j$ if $b_j = \min\{b_i \mid 1 \leq i \leq n\}$
2. For (each) interval I_i, we consider $\mathcal{J}_i$, the set of intervals which overlap with I_i; $right(I_i)$ is the interval of $\mathcal{J}_i$ which ends last. If there is no interval which overlaps with I_i, then the set $\mathcal{J}_i$ is empty, and $right(I_i)$ is not defined.
   Thus if $b_j = \max\{b_k \mid a_k < b_i < b_k\}$ is defined, $right(I_i) = I_j$, otherwise $right(I_i) =$ nil.
3. For (each) interval I_i, we consider $\mathcal{Z}_i$, the set of intervals which begin after the right end-point of I_i; $next(I_i)$ is the interval of $\mathcal{Z}_i$ which ends earliest. If there is no interval which starts after I_i ends, then the set $\mathcal{Z}_i$ is empty, and $next(I_i)$ is not defined.
   Thus, if $b_j = \min\{b_k \mid b_i < a_k\}$ is defined, $next(I_i) = I_j$, else $next(I_i) =$ nil.

It has been shown in Olariu [27] that computation of $right(I_i)$ (for all I_i) involves sorting and prefix-maximum; computation of $next(I_i)$ (for all I_i) involves sorting and suffix-minima. Thus, both the above functions can be computed on 2-d reconfigurable mesh in $O(1)$ time using only $n \times n$ processors. Also, computation of $first(\mathcal{I})$ only involves computation of minimum of n numbers, and hence can be done within the same bounds

2.4.2 Algorithms for interval graphs

Maximum independent set

A subset S of vertices in an interval graph is an *independent set* if the intervals they correspond to are mutually non-overlapping. The *Maximum Independent Set*(MIS) problem is to compute an independent set having maximum cardinality. Golumbic

[14] has shown that there exists a MIS S in $\mathcal{I}$ such that $first(\mathcal{I}) \in S$ and for any $I_i \in S$, $next(I_i) \in S$.

Thus, if a forest is built such that each element I_i points to $next(I_i)$, the MIS consists of exactly those elements which are marked while traversing from $first(\mathcal{I})$ to the root of the tree it belongs to. From the discussion in Section 3.1, the necessary computation can be carried out in $O(1)$ time on a $n \times n$ mesh.

Minimum dominating set

A subset S of vertices in a graph $G = (V, E)$ is a *dominating set* if each vertex of $V - S$ (outside subset S) is adjacent to some vertex of the subset S. The *Minimum Dominating Set*(MDS) problem is to compute a dominating set having minimum cardinality.

The algorithm as in [27] involves construction of a forest where each element I_i points to $right(next(I_i))$(if it exists) or to $next(I_i)$ otherwise. The MDS consists of exactly those elements which are marked while traversing from $right(first(\mathcal{I}))$ (if it exists) or $first(\mathcal{I})$ otherwise, to the root of the tree it belongs to. Again, as evident from previous discussions, the necessary computation can be carried out in $O(1)$ time on a $n \times n$ mesh.

Minimum clique cover

For a graph $G = (V, E)$ *clique cover* is a partition of vertices V of graph G into non-empty disjoint sets such that the vertices belonging to each set form a clique. *Minimum clique cover* (MCC) is a Clique Cover having smallest number of partitions.

In an interval graph, the cardinality of the MCC is the same as the cardinality of the MIS and also the MCC can be easily obtained by a simple Nearest 1 computation once the MIS is available, as shown in [27]. Thus, this algorithm can also be implemented in $O(1)$ time on a $n \times n$ reconfigurable mesh.

2.5 Conclusion

In this chapter we considered fast implementations of interval graph algorithms on the 2-d reconfigurable mesh. Since the reconfigurable mesh looks in some ways similar to a mesh of trees [26], it would be natural to ask about the time and processor complexities of these algorithms on this model. Let us consider the main blocks as laid out: prefix minima computation can be carried out in $O(\log n)$ time using n processors and sorting on a $n \times n$ mesh of trees takes $O(\log n)$ time. Here, the bottleneck operation is the marking of the elements in a linked list which can not be done in even $o(\log^2 n)$ time on a $n \times n$ mesh of trees. However there is some evidence that it is extremely unlikely that there is an $o(\log^2 n)$ time using polynomial processors [30]. Another open problem is to carry out list-ranking on a 2-d $n \times n$ reconfigurable mesh.

Chapter 3

The most vital edge problem

3.1 Introduction

In many network applications, a problem of interest to the network designer is that of finding the edges of the network which are most important- if these edges are removed from the network, the performance of the network will be greatly affected. A *most vital edge* is an edge, which if removed, causes the maximum change(increase) in the cost of the minimum spanning tree ($MST(G)$) of the graph G. More formally it may be defined as follows: let $G = (V, E)$ be a weighted undirected graph with n vertices and m edges; each edge e has a weight $w(e)$ assigned to it. Let $f(G)$ be the weight of a minimum spanning tree of G if G is connected; otherwise $f(G) = \infty$. The most vital edge is an edge e such that $f(G-e) \geq f(G-e')$ for every other edge e' of G. The model of parallel computation used in this chapter is the CREW(Concurrent Read, Exclusive Write) PRAM.

The organization of this chapter is as follows : in Section 3.2 some definitions are briefly reviewed. In Section 3.3 we describe an algorithm which runs on the CREW model in $O(\log n)$ time using $\frac{n^2}{\log n}$ processors; and in Section 3.4 an $O(\log m)$ time, m processor algorithm is described.

3.2 Preliminaries

We may assume that edge-connectivity of G is more than or equal to two as otherwise a bridge can be easily located and reported as the most vital edge. We can further assume without loss of generality that the edge weights are distinct to ensure that G has an unique minimum spanning tree; otherwise we may order the edges in order of occurrence.

If e is an edge of $MST(G)$, and $G-e$ is connected the *replacement edge* $r(e)$ [15] is defined to be the edge such that $MST(G)-e+r(e)$ is a minimum spanning tree of $G-e$. If $G-e$ is not connected $r(e)$ is undefined. As the most vital edge of G belongs to $MST(G)$ [15], for a bridgeless graph G the most vital edge is that edge which maximizes $w(r(e))-w(e)$ among $e \in MST(G)$.

In this chapter, *tree edges* refer to *edges belonging to* $MST(G)$, and *non-tree* the *remaining edges* in the graph. It will also be assumed that the minimum spanning tree $MST(G)$ is available. Minimum spanning tree can be computed on the EREW(and hence CREW) models in $O(\log n \log\log n)$ using $n+m$ processors [20] and on the PRIORITY CRCW model in $O(\log m)$ time using $\frac{(n+m)\log^{(3)} n}{\log n}$ processors [11, 17].

Plane sweep tree[5, 2] T can be defined as follows. Let $S=\{s_1,s_2,...,s_n\}$ be a set of non-intersecting line segments in the plane and T be a complete binary tree with $n+1$ leaves. The leaves of T, in left to right order correspond to intervals $[(-\infty,x_1),(x_1,x_2),...(x_n,+\infty)]$. Associated with each internal node $v \in T$ is an interval I_v which is the union of intervals associated with descendants of v. Let π_v denote the vertical strip $I_v \times (-\infty,+\infty)$. A segment s *covers* a node v if s spans π_v but not $\pi_{parent(v)}$. For each node $v \in T$ let $Cover(v)$ denote the set of segments that cover v. We will use the term Plane Sweep tree to refer to T together with the lists $Cover(v)$ stored at each node. Atallah et. al (*Theorem 5.2* [5]) have shown that the plane sweep tree T can be constructed in $O(\log n)$ time using n processors on the CREW model. At the end of the construction procedure, each node v in T has its list $Cover(v)$ sorted by the "above" relation.

Given a directed graph $G=(V,E)$, in which each node v contains a sorted list

$C(v)$, the fractional cascading problem [10, 5] is to construct a data structure such that given a walk(a sequence of adjacent vertices) $(v_1, v_2, ..., v_n)$ in G and an arbitrary element x, a single processor can quickly locate x in each $C(v_i)$. The term *multilocation* refers to this procedure of locating x in multiple lists.

If $d(G)$ is the maximum degree(indegree/outdegree) of any vertex in G and $Out(v)$ is the set of all nodes $w \in V$ such that $(v, w) \in E$, Atallah et. al (*Theorem 3.10* [5]) have shown that when $d(G)$ is $O(1)$ or $Out(v)$ is given in sorted order for each V, a fractional cascading data structure can be built for G in $O(\log n)$ time using $\frac{n}{\log n}$ processors on the CREW model; here $n = |V| + |E| + \sum_{v \in V} |C(v)|$.

3.3 Optimal algorithm for dense graphs

In this section we describe a simple $O(\log n)$ time algorithm using $\frac{n^2}{\log n}$ processors. The time-processor product matches that of the sequential $O(n^2)$ algorithm of Hsu. et al [15]; these algorithms are optimal for dense graphs.

The input is in an $n \times n$ matrix W; if there is a non-tree edge $e \in G$ of weight $w(e)$ between vertices i, j then $W[i, j] = W[j, i] = w(e)$, else $W[i, j] = \infty$.

```
Step 1 We build an auxiliary matrix C as follows:
    Step 1a Using $n^2$ processors initialize $C[i, j] = \infty$.
    Step 1b For each leaf vertex $i$ pardo
            For all vertices $j$ do
                $C(i, j) = W(i, j)$
Step 2 For each non-leaf vertex $i$ pardo
        For each vertex $j$ do
            Recursively define $C(i, j) = \min\{W(i, j), C(i_1, j), C(i_2, j), ...\}$
            where $i_1, i_2, ...$ are children of $i$.
            Parallel tree contraction technique [1] can be used to compute $C(i, j)$.
```

Step 3 For each tree edge e from i to its parent $p(i)$ in $MST(G)$, let j^* be such that $C(i,j^*) = \min\{C(i,j)$: j is not in the subtree rooted at i $\}$. Define $r(e) = (i,j^*)$.

Step 4 Find the edge $e \in MST(G)$ for which $w(r(e)) - w(e)$ is maximum.

Lemma 3.1 The above algorithm correctly computes $r(e)$ in $O(\log n)$ time with $\frac{n^2}{\log n}$ processors.

Proof: The correctness of this algorithm is obvious from [15]. To obtain the time complexity, we note that Step 2 for a single (fixed) vertex j can be easily done in $O(\log n)$ time with $\frac{n}{\log n}$ processors [1]; thus step 2 takes $O(\log n)$ time with $\frac{n}{\log n} \times n = \frac{n^2}{\log n}$ processors. In Step 3, we assign n processors to each row of C - each processor checks in $O(1)$ time if it is in the subtree rooted at i . All processors which find that they do not match the required condition "remove" themselves and the remaining processors proceed to compute the minimum. Clearly, this step can also be implemented in $O(\log n)$ time with $\frac{n^2}{\log n}$ processors. ∎

3.4 An m-processor algorithm

Though the algorithm in the previous section is optimal for dense graphs, for most applications it is preferable to have an algorithm whose cost is a function of number of edges in the graph, specially if the graph is sparse. In this section, we describe an $O(\log m)$ time algorithm using m processors.

We first compute the pre-order number for all vertices with respect to $MST(G)$; this can be done by Euler tour traversal technique [31] in $O(\log n)$ time with $\frac{n}{\log n}$ processors. Using the Euler tour, the number of descendants of each vertex in $MST(G)$ can also be computed within the same resource bounds. We let $pre(v)$ denote the pre-order number for vertex v and $des(v)$ the number of descendants of vertex v, respectively .

To compute the most vital edge, essentially, for each tree edge e, we need to compute its replacement edge $r(e)$. Let us consider a tree edge $e = (a,b)$ where $pre(a) < pre(b)$. As one end-point of $r(e)$ is necessarily a descendant of b while the other end-point is a non-descendant, the replacement edge $r(e)$ is the minimum cost edge in the set of edges having one end-vertex in subtree rooted at b and the other end-vertex is not in the sub-tree rooted at b.

We represent each non-tree edge (a,b), $pre(a) < pre(b)$ as a horizontal segment in the plane with x-coordinate ranging from a to b, i.e segment $\{(pre(a), pre(b))\}$. The y-coordinate of the segment is defined as follows: if there are k segments having first x-coordinate a, the i'th segment is assigned a y-coordinate $y_i(a) = a + \frac{i-1}{k}$. The cost of each non-tree edge is assigned to its corresponding segment. Note that by definition, all the segments are strictly ordered on their y-coordinate.

Now, any descendant v_i of a given vertex v, has a pre-order number between $pre(v)$ and $\alpha_v = pre(v) + des(v)$, i.e $pre(v) \leq pre(v_i) \leq \alpha_v$. An edge starting from a descendant of v is represented by a segment whose first x-coordinate x' fulfills the condition $pre(v) \leq x' \leq \alpha_v$ and an edge ending at a descendant of v is represented by a segment whose second x-coordinate x'' lies between $pre(v)$ and α_v i.e $pre(v) \leq x'' \leq \alpha_v$.

Thus, we can solve the problem if we can find for each vertex v the least cost segment among those which satisfy *any* one of the following conditions:

1. The segment starts at j_1, $(pre(v) \leq j_1 \leq pre(v) + des(v))$ and ends at j_2, where $j_2 \geq pre(v) + des(v) + 1$.
2. The segment starts at j_1', $(1 \leq j_1' < pre(v))$ and terminates at j_2', $(pre(v) \leq j_2' \leq pre(v) + des(v))$.

The two cases are similar(they are in fact equivalent : case (2) viewed from the right is exactly same as case (1)). In subsequent discussions we will only consider case (1).

As the set of segments is strictly ordered, a plane-sweep tree[5] T can be constructed for this set. For each node v in T we consider $Cover(v)$, the set of all segments that *cover* v. By construction of T the segments belonging to $Cover(v)$ are sorted in increasing order of their y-coordinates. This implies that segments starting from different vertices in G are arranged in increasing order of their x-coordinates while segments starting from the same vertex appear consecutively in $Cover(v)$.

Now, for vertex v let E_v be the leaf in T associated with the integer α_v+1. By the property of plane-sweep tree if we consider the leaf-to-root walk $(E_v, v_1, v_2 ..., root(T))$ in T, the set $Cover(E_v) \bigcup Cover(v_1) \bigcup Cover(v_2) \bigcup ... \bigcup Cover(root(T))$ contains exactly those segments that start before the vertical line $l_v = \alpha_v + 1$ and end after l_v. Thus our desired query is to locate the least-cost segment which belongs to this set and starts after(or at) $pre(v)$.

We can compute the least-cost segment starting after(or at) $pre(v)$ for a node v_i belonging to T by a suffix minima(on key $cost$) on the list $Cover(v_i)$ followed by a binary search to find the first segment whose starting x-coordinate is greater than or equal to $pre(v)$. Obviously, due to the suffix-minima computation, this segment can provide the answer to the desired query. To increase the efficiency(the number of elements in the sorted list) of the binary search, we remove from $Cover(v)$ sets of segments which have the same starting x-coordinate and retain only the segment with minimum cost(after suffix computation) of each set. Naturally, the first segment of each set happens to satisfy this condition; so it is sufficient to retain only the first segment of each set.

Thus, for a single node v_i in the walk, the query can be answered by a single processor in $O(\log m)$ time, as $|Cover(v_i)| \leq m$, assuming the suffix-computation has been performed. As the length of the walk is $O(\log m)$, the overall query can be answered in $O(\log^2 m)$ time by a single processor.

To answer the query faster, i.e in $O(\log m)$ time, we convert T into a fractional cascading data structure T' [5]. The query can be answered in the manner described below. In $Cover(E_v)$ we find the first segment that has a x-coordinate greater than or equal to $pre(v)$. Since suffix minima has been computed earlier, this gives us the minimum cost segment among those belonging to $Cover(E_v)$ and starting after(or at) $pre(v)$. We then perform multilocation along the leaf-to-root walk, i.e from E_v to $root(T')$. From the property of T and T', only $O(1)$ time needs to be spent for each node along this walk. Since the length of the walk is $O(\log m)$, answering each query needs $O(\log m)$ time.

Theorem 3.1 The above algorithm correctly computes $r(e)$ in $O(\log m)$ time with

m processors.

Proof: The correctness of the algorithm follows from its construction. T can be constructed in $O(\log m)$ time with m processors [5]. For the suffix-minima step, since there are $O(m \log m)$ elements overall in T, this step again needs $O(\log m)$ time with m processors. T' is also constructible within the same bounds. As we have n queries each of which can be answered independently in $O(\log m)$ time, the query part uses n processors and takes $O(\log m)$ time. Thus, the theorem follows. ∎

3.5 Final remarks

In this chapter, $O(\log m)$ time algorithm using m processors on CREW model has been described. However if an efficient algorithm for the *maximum spanning tree* can be designed, then, using an observation of Iwano and Katoh[18] the number of non-tree edges to be considered in the algorithm can be reduced to n only and hence the number of processors can also be reduced to $O(n)$.

Chapter 4

DFS and BFS in permutation graphs

4.1 Introduction

Let π be a permutation of n integers in range $[1..n]$. For this permutation, a graph $G(\pi)$ is defined in which there is a vertex for each integer i and an edge between integers i and j if $(i-j)(\pi^{-1}(i)-\pi^{-1}(j))<0$ where $\pi^{-1}(i)$ denotes the position of integer i in π. An undirected graph $G=(V,E)$ is called a permutation graph if there exists a π such that G is isomorphic to $G(\pi)$ [14]. Thus, for a permutation graph we use the terms "vertex" i and "integer" i interchangeably in rest of the chapter.

A DFS tree for a (general) graph $G=(V,E)$ is a spanning tree such that if any two vertices have an edge in the graph, one of them is an ancestor of the other (in this tree). A BFS tree is a spanning tree such that if any two vertices have an edge in the graph, their levels in the tree do not differ by more than 1. For (general) graphs $G=(V,E)$, the problems of constructing BFS and DFS trees have simple $O(n+m)$ time sequential solutions, where $n=\mid V\mid$ and $m=\mid E\mid$. The problem of efficient (optimal) parallel construction of a BFS or a DFS tree for a general graph is open.

In this chapter, we consider the problems of finding a DFS tree and a BFS tree of a permutation graph provided the permutation π is given. Under this assumption, the best sequential algorithms for constructing a DFS takes $O(\min(n\log\log n, n+$

m)) and for BFS tree $O(n)$ time [29]. The proposed parallel solution for the DFS problem is based on finding longest increasing subsequence in parallel. This problem has a $O(n \log \log n)$ time sequential solution, but the best parallel solution requires $O(\log^3 n)$ time with n processors on a Concurrent Read Exclusive Write (CREW) PRAM model of computation [23] and $n\frac{(\log \log n)^2}{\log n}$ processors on an ARBITRARY-WRITE Concurrent Read Concurrent Write (CRCW) PRAM [28]. On a CREW PRAM, more than one processor can simultaneously read from the same memory location, but concurrent writes are not allowed. An ARBITRARY-WRITE CRCW PRAM model allows concurrent writes with the stipulation that in case of multiple writes in a single memory location, an arbitrary processor succeeds. The solution for the BFS problem takes $O(\log n)$ time using $\frac{n}{\log n}$ processors on an Exclusive Read Exclusive Write (EREW) PRAM. In this model, no two processors can simultaneously access the same memory location either for reading or for writing.

The organisation of this chapter is as follows: in Section 4.2 we consider the problem of constructing a DFS tree and in Section 4.3 the problem of constructing a BFS tree.

4.2 DFS tree

The proposed parallel algorithm is obtained by parallelizing a slightly modified version of sequential algorithm of Rhee et.al. [29]. The organisation of this section is as follows: first some preliminary definitions and results are discussed, the algorithm is then described and finally resource (processor and time) requirements of the algorithm are analysed.

Preliminaries

A minimum clique cover (MCC) of a graph is a partition of vertices of the graph into a set of cliques, such that the number of cliques is minimum. Since permutation graphs are a subclass of the family of perfect graphs, the cardinality of the MCC is the same as the cardinality of the maximum independent set (MIS) [14].

It is well-known that in a permutation graph, a clique is a decreasing sequence of integers and an independent set is an increasing sequence of integers (see e.g. [14]). A MCC, $CC(\pi) = (C_1, C_2, \ldots, C_q)$ where $q = \mid MCC(\pi) \mid$ of a permutation graph can be obtained in following manner[29]: to obtain all elements C_i^j belonging to clique C_i, all elements belonging to all C_j, $1 \le j \le (i-1)$ are removed from π; the first element left in π is the first element (C_i^1) of C_i and subsequently each element C_i^j in C_i is the first element (going from left to right) less than C_i^{j-1} in reduced π. As is obvious, each C_i is a maximal clique and the C_i's constitute a clique cover.

Thus, the vertices of each clique C_i are arranged in descending order; if $l(i)$ denotes the leftmost (largest) element in C_i and $r(i)$ denotes the rightmost (smallest) element in C_i, the C_i's are indexed such that the $r(i)$'s are arranged in ascending order.

For clique C_i, we consider all cliques C_j, $(j < i)$ where C_j contains a vertex k such that $k < l(i)$ and k has an edge with $l(i)$ or, occurs after $l(i)$ in π. The clique $C_l(i)$ is defined to be the clique with maximum index satisfying this condition, thus if $t^* = \max[t \mid C_t \text{ contains } k, k < l(i) \text{ and } \pi^{-1}(k) > \pi^{-1}(l(i))]$ then $C_l(i) = C_{t^*}$. The element $l\text{-}adj(i)$ is the largest value in $C_l(i)$ which satisfies the given condition, i.e, if $k^* = \max[k \mid k \in C_l(i) \text{ and } k < l(i) \text{ and } \pi^{-1}(k) > \pi^{-1}(l(i))]$ then $l\text{-}adj(i) = k^*$.

Next, for the clique C_i we consider all cliques C_j, $(j < i)$ where C_j contains a vertex k such that $k > r(i)$ and k has an edge with $r(i)$ or, occurs before $r(i)$ in π. The clique $C_r(i)$ is defined to be the clique with maximum index satisfying this condition, i.e, if

$t^* = \max[t \mid C_t \text{ contains } k, k > r(i) \text{ and } \pi^{-1}(k) < \pi^{-1}(r(i))]$ then $C_r(i) = C_{t^*}$. The element $r\text{-}adj(i)$ is the smallest value in $C_r(i)$ which satisfies the given condition, i.e if

$k^* = \min[k \mid k \in C_r(i) \text{ and } k > r(i) \text{ and } \pi^{-1}(k) < \pi^{-1}(r(i))]$ then $r\text{-}adj(i) = k^*$.

Algorithm

We next describe the algorithm for computing a DFS tree. All elements of a clique are to be oriented in the same direction to avoid cross-edges. So, in some sense each clique tries to choose the clique nearest to it and the elements set up their parent pointers accordingly.

Step 1 Compute $CC(\pi) = (C_1, C_2, \ldots, C_q)$, where the C_i's are indexed as discussed earlier.

REMARK: In this step, the minimum clique cover is computed and the elements in the cliques are arranged as defined earlier.

Step 2 For each clique C_i compute $C_l(i)$, $l\text{-}adj(i)$, $C_r(i)$, $r\text{-}adj(i)$. Define $C_s(i)$ to be the clique with larger index among $C_l(i)$ and $C_r(i)$.

REMARK: In this step, each clique C_i tries to choose the clique $C_s(i)$ nearest to it. Intuitively, consider cliques C_j, C_k, with $j, k < i$; further assume that $k < j$. Now, if the elements in C_i have edges with elements in both C_j and C_k, while elements in C_j also have edges with elements in C_k, then C_i chooses C_j and C_j chooses C_k as the nearest clique to it so that we can have the DFS traversal with C_k = parent(C_j) and C_j = parent(C_i).

Step 3 We define the parent $p(C_i^j)$ of each element C_i^j in the DFS tree as follows:

```
$p(C_1^1) = \phi$. /* First vertex of first clique is the root */
For each element $j$ in $C_1$ do    $p(C_1^j) = C_1^{j-1}$.
/* In first clique, parent of each element is the previous element. This is
subsequently referred to as the backward direction. The forward direction
correspondingly implies that the parent of each element is the next element. */
For each element $j$ in $C_i$ do /* Find parent of vertices in $i$'th clique */
if $l\text{-}adj(i) \in C_s(i)$ then /* i.e, if $C_s(i) = C_l(i)$ */
    $p(C_i^j) = C_i^{j-1}$. /* $C_i$ is oriented in backward direction */
    if $p(C_s^1(i)) = C_s^2(i)$ then /*Nearest clique is oriented in forward direction */
        $p(l(i)) = l\text{-}adj(i)$.
    else /* Nearest clique is oriented in backward direction */
        $p(l(i)) = r(s)$.
/* The parent of $l(i)$ is the last element of the nearest clique $C_s(i)$. */
else /* $C_s(i) = C_r(i)$ */
    $p(C_i^j) = C_i^{j+1}$. /* $C_i$ is oriented in forward direction */
    if $p(C_s^1(i)) = C_s^2(i)$ then /*Nearest clique is oriented in forward direction */
        $p(r(i)) = l(s)$.
  /* The parent of $r(i)$ is the first element of the nearest clique $C_s(i)$. */
    else /* Nearest clique is oriented in backward direction */
        $p(r(i)) = r\text{-}adj(i)$.
```

Initially, the elements of C_1 are laid out in a straight chain with the maximum element (in value) as the head of the chain. Obviously, this is a DFS traversal for elements belonging to C_1. We refer to such a chain as a backward chain and a chain in the reverse direction as a forward chain. Subsequently, members of clique C_i check whether its nearest clique is oriented in the forward or backward direction. The condition $p(l(C_s(i))) \in C_s(i)$ is true implies that the nearest clique is oriented forward, otherwise, it is oriented in the backward direction. Once the orientation of the nearest clique is known, the exact relationship between C_i and $C_s(i)$ is checked.

For example, let $C_s(i)$ be oriented forward and $l\text{-}adj(i) \in C_s(i)$; $l(i)$ has an edge with all elements belonging to $C_s(i)$ and less than $l\text{-}adj(i)$. This $l(i)$ cannot be hooked to any element k less than $l\text{-}adj(i)$, as it would result in cross-edges ($l(i)$ has an edge with $l\text{-}adj(i)$ in the graph but is neither an ancestor nor a child of $l\text{-}adj(i)$ in the DFS tree). Thus $l(i)$ is hooked to $l\text{-}adj(i)$.

Lemma 4.1 The above algorithm correctly computes the DFS tree for the given graph.

Proof: See [29]. ∎

Lemma 4.2 Step 1 can be implemented in $O(\log^3 n)$ time with n processors on a CREW PRAM or in $O(\log^3 n)$ time with $n\frac{(\log\log n)^2}{\log n}$ processors on an ARBITRARY write CRCW PRAM.

Proof: Computing the maximum cardinality independent set (MIS) in a permutation graph is equivalent to computing the longest increasing subsequence (LIS) of π and can be done by finding the longest common subsequence (LCS) of two strings, one of

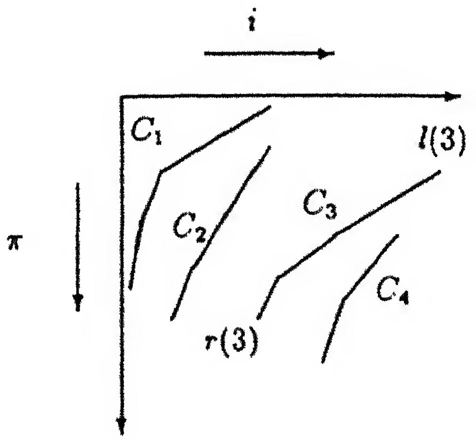

Fig 1

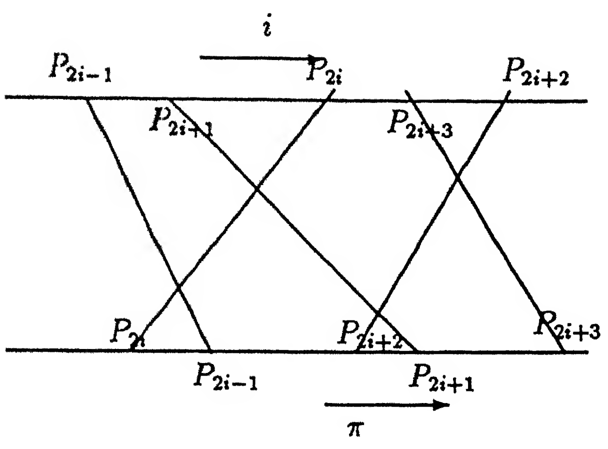

Fig 2

them π and the other the sequence of the n integers $1 \dots n$. This computation can be performed by Lu's algorithm [23, 28] within the stated resource bounds.

In the LIS computation of Lu's algorithm, the *class* of each point is also computed. A point (an integer) p belongs to a *class* i if the longest increasing subsequence in $\pi[1..\pi^{-1}(p)]$ is of length i. If elements belonging to all classes j, $(1 \leq j < i)$ are removed, then the length of longest increasing subsequence (in reduced π) for each element belonging to class i is 1, i.e the elements belonging to class i form a decreasing subsequence in π. Hence, the classes partition the integers in π into decreasing subsequences; as each class is a decreasing subsequence in π it is a clique in the graph. Also, the number of classes is equal to the cardinality of the MIS. Thus, the classes form a *minimum clique cover.* Observe that the value of $r(i)$, the least integer belonging to class i, is more than the value of $r(j)$ for class j, $(j < i)$; otherwise, the element $r(i)$ would occur in some class less than or equal to j. Thus, the i'th class is exactly C_i as defined. In Fig 1, corresponding to some permutation π the classes (cliques) computed by Lu's algorithm are shown.

From the above discussion, the resource bounds for obtaining the minimal clique cover $CC(\pi)$ required in the algorithm are the same as those used in algorithm for finding an MIS. ∎

Lemma 4.3 Step 2 can be implemented on an EREW PRAM in $O(\log n)$ time using $\frac{n}{\log n}$ processors.

Proof: We discuss algorithm for computation of *l-adj*(i); computation for *r-adj*(i) is similar. An array W is constructed in following manner: the positions of elements in C_i occur in successive locations of W in ascending order, ($r(i)$ occurs first and $l(i)$ last). The location just before the location for element $r(i)$ in W contains another copy of the value of $l(i)$. Thus, if $| C_i | = x$, C_i occupies $x+1$ locations in W. Also, the ordering between C_i's is maintained, i.e C_j, $j < i$ occurs before C_i in W. We now claim that a nearest larger [8] computation in W solves the problem.

To prove this, recall that the elements in each clique C_i are arranged in order both by value as well as position in π. Now, if $C_l(i) = C_j$, $j \leq i$, all elements in all cliques C_k $(j < k < i)$ have positions less than $l(i)$. So, none of the $l(k)$'s of such cliques can be the nearest larger for element $l(i)$. Now, searching from $l(i)$ towards the left in W, once we reach $C_l(i)$, we know that the elements are arranged in increasing order (as seen from $l(i)$). Thus, the first element that $l(i)$ sees larger than itself is *l-adj*(i). The array W can be constructed optimally in $O(\log n)$ time with linear work on an EREW PRAM and a nearest larger computation can also be carried out within the same bounds. Once *l-adj*(i) has been computed, the identity of $C_l(i)$ is also known. ∎

Lemma 4.4 Step 3 can be implemented on an EREW PRAM in $O(\log n)$ time with $\frac{n}{\log n}$ processors.

Proof: It is obvious from the algorithm that necessary information for all elements C_i^j of a clique C_i can be computed in $O(1)$ time from $C_s(i)$. We therefore construct a tree in which there is a node for each clique C_i and the parent of each C_i is $C_s(i)$. So, once the desired assignments are made to all elements of C_1, we just need a top-down algebraic tree contraction step (T-ATC) [1] to compute the information for all elements present in nodes of tree. This step can therefore be implemented in $O(\log n)$ time using $\frac{n}{\log n}$ processors on an EREW PRAM. ∎

Theorem 4.1 DFS tree of a connected permutation graph can be constructed in $O(\log^3 n)$ time with n processors on a CREW PRAM, or alternatively in $O(\log^3 n)$ time with $n\frac{(\log\log n)^2}{\log n}$ processors on an ARBITRARY CRCW PRAM.

Proof: The proof of correctness follows from Lemma 4.1 and the resource bounds required for the computation follow from Lemmas 4.2, 4.3 and 4.4. ▮

4.3 BFS tree

Preliminaries

The *matching diagram* [14] for a permutation graph consists of two rows of integers. The upper row, U is the sequence of all integers in the range $[1..n]$ while the lower row D is the given permutation, π. Each integer i in U is connected by a straight line to that position j in D such that $\pi(j) = i$; thus we may view the graph as a bijective mapping of integers from U to D; an edge (i, j) is in the graph if and only if the lines i and j intersect in the matching diagram.

In the matching diagram the elements which are tilted towards the right, i.e integers k such that $\pi^{-1}(k) > k$ are called R-elements. Similarly, elements tilted towards the left, i.e, integers k, $\pi^{-1}(k) < k$ are labelled as L-elements.

Observation 4.1 In a permutation graph, for each R-element r there exists at least one L-element l which intersects it such that $l \geq \pi^{-1}(r)$.

Proof: Both U and D are permutations of n integers. If there does not exist a L-element l which intersects r, this implies that $n - r$ integers in U are mapped to $n - \pi^{-1}(r) < n - r$ integers in D, which is a contradiction. Again, if $l < \pi^{-1}(r)$ for all L-elements l intersecting line r, let l' be maximum among all such l's. But, $n - \pi^{-1}(r)$ integers in D would then map to $n - l' > n - \pi^{-1}(r)$ integers in U. Thus, there is a contradiction. ▮

Observation 4.2 In a connected permutation graph, for each R-element r such that $\pi^{-1}(r) < n$, there exists at least one L-element l intersecting it such that $l > \pi^{-1}(r)$.

Proof: In observation 1 we have proved that there exists l where $l \geq \pi^{-1}(r)$ for any R-element. If $l = \pi^{-1}(r)$, then $n - l$ integers in U map exactly to $n - \pi^{-1}(r) = n - l$ integers in D, i.e the integers in U in the range $l + 1$ to n are in a separate connected component than the one in which integer l lies; thus there is a contradiction. ▮

b) if $\pi^{-1}(P_{2i-1}) < \pi^{-1}(j) < \pi^{-1}(P_{2i+1})$ then $p(j) = P_{2i+1}$; $level(j) = 2i+2$.

/* If the first odd-indexed element whose position is less than position of j in D is P_{2i-1}, the parent of j in the BFS tree is P_{2i+1}.*/

c) else /* The first odd-indexed element whose position is less than position of j in D is P_{2i+1}. */

$p(j) = P_{2i+2}$; $level(j) = 2i+3$.

/* If this condition holds the parent of j in the BFS tree is P_{2i+2}*/

REMARK: Effectively, in this step each element j has to locate its corresponding P_{2i} i.e, the first even-indexed element in the marked path such that $P_{2i} < j$. Once this element is located, the values of P_{2i+1}, P_{2i-1}, etc, which are needed in (b) and (c) are also available. This is achieved by marking the values of the even elements (in the marked path) in U. Each non-marked element needs to locate the nearest marked element to its left. This can be done in $O(\log n)$ time with $\frac{n}{\log n}$ processors on an EREW PRAM.

For proof of various lemmas below, we refer to the matching diagram of Fig 2.

Lemma 4.5 For each element j, the parent $p(j)$ is an element with which j has an edge in the graph.

Proof: If $j = P_i$ for some i, then by definition, $p(j)$ has an edge with j. Otherwise, we consider $p(i)$ as defined in cases (b) and (c) of the algorithm. If case (b) holds, we note that $\pi^{-1}(j) < \pi^{-1}(P_{2i+1})$ and $j > P_{2i}$ but $P_{2i} > P_{2i+1}$, i.e $j > P_{2i+1}$, or, there is an edge between j and $p(j)$ in the original graph. A similar treatment is valid for case (c). ∎

Lemma 4.6 For each element j, the parent pointers $p(j)$ define a path to the root of the tree.

Proof: Obvious from construction. ∎

Lemma 4.7 For any two elements j and k in the graph, $| level(j) - level(k) | \leq 1$.

b) if $\pi^{-1}(P_{2i-1}) < \pi^{-1}(j) < \pi^{-1}(P_{2i+1})$ then $p(j) = P_{2i+1}$; $level(j) = 2i + 2$.

/* If the first odd-indexed element whose position is less than position of j in D is P_{2i-1}, the parent of j in the BFS tree is P_{2i+1}.*/

c) else /* The first odd-indexed element whose position is less than position of j in D is P_{2i+1}. */

$p(j) = P_{2i+2}$; $level(j) = 2i + 3$.

/* If this condition holds the parent of j in the BFS tree is P_{2i+2}*/

REMARK: Effectively, in this step each element j has to locate its corresponding P_{2i} i.e, the first even-indexed element in the marked path such that $P_{2i} < j$. Once this element is located, the values of P_{2i+1}, P_{2i-1}, etc, which are needed in (b) and (c) are also available. This is achieved by marking the values of the even elements (in the marked path) in U. Each non-marked element needs to locate the nearest marked element to its left. This can be done in $O(\log n)$ time with $\frac{n}{\log n}$ processors on an EREW PRAM.

For proof of various lemmas below, we refer to the matching diagram of Fig 2.

Lemma 4.5 For each element j, the parent $p(j)$ is an element with which j has an edge in the graph.

Proof: If $j = P_i$ for some i, then by definition, $p(j)$ has an edge with j. Otherwise, we consider $p(i)$ as defined in cases (b) and (c) of the algorithm. If case (b) holds, we note that $\pi^{-1}(j) < \pi^{-1}(P_{2i+1})$ and $j > P_{2i}$ but $P_{2i} > P_{2i+1}$, i.e $j > P_{2i+1}$, or, there is an edge between j and $p(j)$ in the original graph. A similar treatment is valid for case (c). ∎

Lemma 4.6 For each element j, the parent pointers $p(j)$ define a path to the root of the tree.

Proof: Obvious from construction. ∎

Lemma 4.7 For any two elements j and k in the graph, $| level(j) - level(k) | \leq 1$.

Proof: We consider the set of elements defined by case (b) of the algorithm. This set of elements has P_{2i+1}, which is a R-element, as parent, and we denote this set as R_{2i+1}.

Note that for any element $k \in R_{2i+1}$, $\pi^{-1}(k) > P_{2i-1}$ which is a R-element. If $\pi^{-1}(k) < P_{2i-1}$, then instead of P_{2i-1} choosing $p_m(P_{2i-1}) = P_{2i}$, it would have chosen $p_m(P_{2i-1}) = k$, i.e there would be a contradiction.

Thus, the R-sets constitute a partition, or no element in a R-set can have an edge with an element in another R-set.

Similarly, we consider the set of elements defined as case (c) with parent P_{2i+2} and denote this set as L_{2i+2} as this set is associated with a L-element. For an element $k' \in L_{2i+2}$, $\pi^{-1}(k') < P_{2i+3}$ as there would be a contradiction otherwise. The L-sets constitute a partition similar to the R-sets.

We now consider an element $j \in R_{2i+1}$. As already seen, it may have edges only with elements k belonging to any of the three sets: (i) R_{2i+1} (ii) L_{2i} (iii) L_{2i+2}. If case (i) holds, obviously $level(j) = level(k)$. For cases (ii) and (iii), we recall that $level(P_{2i}) < level(P_{2i+1}) < level(P_{2i+2})$. So, if case (ii) holds, $level(k) = 2i+1$, and for case (iii), $level(k) = 2i+3$. But $level(j) = 2i+2$. Thus, $\mid level(j) - level(k) \mid \leq 1$ for all edges (j,k) in the original graph. ∎

Theorem 4.2 The BFS tree of a connected permutation graph can be computed in $O(\log n)$ time with $\frac{n}{\log n}$ processors on an EREW PRAM.

Proof: The correctness of the algorithm follows from Lemmas 4.5, 4.6 and 4.7. Lemmas 4.5 and 4.6 imply that the $p(i)$'s define a spanning tree and Lemma 4.7 proves that this tree is indeed a BFST. ∎

4.4 Conclusion

In this chapter, an optimal parallel algorithm for constructing the BFS tree of a permutation graph has been described. The algorithm for the DFS tree is constrained by the resource requirements for constructing the MCC (MIS) of a permutation graph. The sequential complexity of the algorithm for the MCC problem is $O(n \log\log n)$.

However, as other steps of the DFS algorithm take only $O(\log n)$ time using $\frac{n}{\log n}$ processors on an EREW PRAM, discovery of a better algorithm for MCC problem will result in corresponding improvement in our algorithm.

Postscript

Very recently Ibarra and Zheng [16] have discovered a $O(\log n)$ time optimal algorithm for shortest paths from any source node on an EREW PRAM in an unweighted permutation graph. However, their algorithm does not actually find a shortest path tree or BFS tree.

Chapter 5

Connected and Biconnected Components of a Permutation Graph

5.1 Introduction

We consider a permutation π of n integers in range $[1..n]$. On this permutation, a graph $G(\pi)$ may be defined such that integers i and j have an edge in $G(\pi)$ iff $(i-j)(\pi^{-1}(i)-\pi^{-1}(j)) < 0$ where $\pi^{-1}(i)$ denotes the position of the integer i in π. An undirected graph $G = (V, E)$ is defined to be a permutation graph if there exists a π such that G is isomorphic to $G(\pi)$ [14].

Throughout this chapter, it will be assumed that the permutation graph $G = (V, E)$ has been provided to us in the form of the permutation π. Based on this fact, the terms *vertex* i and *integer* i are used interchangeably. Two common representations of a permutation graph given in this form are the *matching diagram* and the representation of vertices of the graph as points in the plane. In the matching diagram we have an upper row U which is the sequence of all integers in the range $[1..n]$ and a lower row D which is the permutation π. Each integer i in U is joined to the position in D where i occurs and, this line also subsequently refers to vertex i. Two integers i and j have an edge in G iff their corresponding lines intersect. The matching diagram may be viewed as a bijective mapping between integers in $[1..n]$ to their positions in π.

In the plane, each vertex i may be represented by the co-ordinate pair $(\pi^{-1}(i), i)$, i.e the y-coordinate is the value of the integer, while the x-coordinate is its position in π. We also refer to this as the point i. From the definition of π point i has an edge in G with all points which lie in the south-east and north-west quadrants relative to its location in the plane, i.e point $p = (x_p, y_p)$ has an edge with point $q = (x_q, y_q)$ if either (a) $x_p < x_q$ and $y_p > y_q$ or (b) $x_p > x_q$ and $y_p < y_q$.

In this chapter, we obtain a parallel algorithm to find the biconnected components of a permutation graph in $O(\log\log\log n)$ time with linear work on the COMMON(TOLERANT) CRCW PRAM or $O(\log n)$ time with linear work on the CREW PRAM. This improves on the previous $O(\log n)$ time, n^2 processor biconnectivity algorithm on the CREW PRAM of [22]. We also obtain an algorithm for finding the connected components which runs in $O(\log n)$ time with linear work on the CREW PRAM, $O(\log\log\log n)$ time using $\frac{n}{\log\log\log n}$ processors on the COMMON(TOLERANT) CRCW PRAM and $O(\log^* n)$ time with $\frac{n}{\log^* n}$ processors on a Priority-Write CRCW PRAM.

The following routines are referred to frequently in the rest of the chapter:

1. Prefix-maxima(minima):- This routine computes the prefix-maxima(minima) for an array $(a_1, a_2, ... a_n)$ where each a_i is an integer in the range $[1..s]$. This routine takes $O(\log\log\log s)$ time using $\frac{n}{\log\log\log s}$ processors on the COMMON(TOLERAN CRCW PRAM [7] and $O(\log n)$ time using $\frac{n}{\log n}$ processors on the CREW PRAM. Subsequently, we refer to this time as $\tau(n)$ and the processor count as $\rho(n)$. For the special case when $s = n^{O(1)}$ this routine takes $O(\log^* n)$ time [1] with $\frac{n}{\log^* n}$ processors on a Priority-Write CRCW PRAM [7].
2. Merge:- Given two sorted lists $(a_1, a_2, ... a_n)$ and $(b_1, b_2, ... b_n)$, where each $a_i(b_i)$ is an integer in the range $[1..s]$, this routine merges the two arrays to form a sorted array $(c_1, c_2, ..., c_{2n})$. This routine runs in $O(\log\log\log s)$ time using $\frac{n}{\log\log\log s}$ processors on the CREW PRAM [9].

[1] $\log^i n = \log\log^{i-1} n; \log^* n = \min\{j : \log^j n \leq 1.\}$

5.2 Connected components

Lemma 5.1 A connected component in a permutation graph is a consecutive sequence of integers; thus a connected component of size k starting at vertex(integer) l contains all vertices in the range $[l..(l+k)]$.

Proof: We consider a connected component of size k in which the minimum vertex(integer) present is l. Let j, $(l < j < l+k)$ be the first integer not present in the given connected component. We consider the remaining $(k-(j-l))$ vertices in the graph which are greater than j and occur in the same connected component as l. At least one of these vertices has to have an edge with(intersect in the matching diagram) some vertex in the range $[l..(j-1)]$ if all of them lie in the same connected component. But, for this to happen, some vertex in one of the two sets necessarily intersects(has an edge with) the line j, and therefore, j also lies in the same connected component. ∎

Thus, partitioning the vertices of a permutation graph into its connected components is equivalent to identifying a set of integers each of which is the first(least) member of a connected component. We next state the necessary and sufficient condition for integer i to start a connected component.

Lemma 5.2 An integer i starts a connected component iff there does not exist any integer j such that $j < i$ and $\pi^{-1}(j) \geq i$.

Proof: We consider the integers i and j, $(j < i)$ in the upper row U of the matching diagram. Now, if j maps to a position in D which is greater than or equal to i, this implies that $\exists k$ such that $\pi^{-1}(k) < i$ and $k > i$. This follows from the fact that if the $i-1$ integers in the range $[1..(i-1)]$ in U map to i positions in D, there exists at least one integer k in D, $\pi^{-1}(k) < i$ which can not be mapped to an element in U in the range $[1..(i-1)]$. Thus, integer k in L necessarily maps to a position greater than the position of i in U, or k intersects i. In the graph we thus have edges (j,k) and (k,i) where $j < i < k$ and so, i belongs to a connected component whose starting element is less than or equal to j, which is a contradiction. ∎

Lemma 5.3 The integers s_i which start a connected component can be computed in $O(\log n)$ time on the CREW PRAM, or $O(\log\log\log n)$ time on the COMMON CRCW PRAM or $O(\log^* n)$ time on a Priority-write CRCW PRAM with linear work.

Proof: From Lemma 5.2. for all integers to check the condition as stated, a suffix–minima computation is required on array π. Integer s_i starts a connected component if the suffix-minimum value at position s_i is equal to s_i. Each element in π is in the range $[1..n]$ and so, the faster($O(\log^* n)$ time) prefix routine can also be applied. ∎

Lemma 5.4 Each element identifies the connected component it belongs to within the resource bounds of prefix-maxima routine.

Proof: We consider the array N which is the sequence of integers $[1..n]$. Each element which knows that it is the starting element of a connected component, marks itself as 1 and others mark themselves as 0. From Lemma 5.1, each 0 identifies itself in the connected component started by the nearest 1 to its left. ∎

Theorem 5.1 The connected components of a permutation graph can be identified in $O(\log^* n)$ optimal time on a Priority-write CRCW PRAM or, $O(\log\log\log n)$ optimal time on the COMMON(TOLERANT) CRCW PRAM or, $O(\log n)$ optimal time on the CREW PRAM.

Proof: The correctness of the algorithm follows from Lemma 5.1 and 5.2. The resource bounds follow from Lemmas 5.3 and 5.4. ∎

5.3 Biconnected components

In this section we consider the representation of each graph vertex as a point in the plane. A point $p_i = (x_i, y_i)$ is said to be *dominated* by another point $p_j = (x_j, y_j)$ if $x_j < x_i$ and $y_j > y_i$. In the context of a permutation graph, there is an edge between integers y_i and y_j in the graph. The set of *roots* $r_i \in G$ is defined to be the set of elements such that r_i is not dominated by any other point in G, i.e, there does not exist any element j for which $j > r_i$ and $\pi^{-1}(j) < \pi^{-1}(r_i)$.

Lemma 5.5 The set of all r_i's can be computed in $\tau(n)$ time with $\rho(n)$ processors.

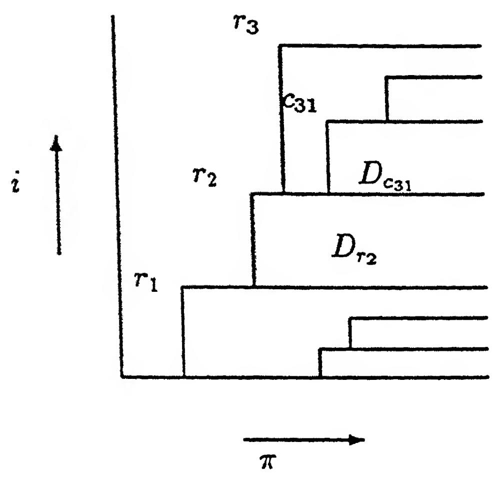

Fig 1: Plane representation

Proof: From the definition of r_i, it follows that in π, r_i does not have any larger element to the left, i.e, r_i's are exactly those elements such that the prefix-maximum(in π) at the location $\pi^{-1}(r_i)$ equals r_i. ∎

Lemma 5.6 In the array π^{-1} the values present in locations $(r_1, r_2 ...)$ form an increasing sequence.

Proof: From the definition of r_i it follows that there is no integer less than r_i and having a position greater than r_i in π. Equivalently, in π^{-1}, no location less than r_i contains a value greater than the value contained in location r_i. ∎

Thus, by construction, S, the set of r_i's, is implicitly ordered both by value and position in π^{-1}.

For any point $p_j = (x_j, y_j)$, we define the *closest dominating root* as that r_i which dominates p_j and is the nearest in y-coordinate among all dominating r_i's. The set of all points which have r_i as their closest dominating root is termed as D_{r_i}. This region is shown in Fig 1.

Lemma 5.7 The set D_{r_i} consists exactly of all the integers in the range $[y_{r_{i-1}} .. y_{r_i}]$.

Proof: By definition, the set D_{r_i} can not have integers with value greater than or equal to y_{r_i}. But if any integer in D_{r_i} is less than $y_{r_{i-1}}$, it is also dominated by r_{i-1}. So, it would choose some r_j, $j \leq (i-1)$ as its closest dominating root. Thus, only integers k, $(y_{r_{i-1}} < k < y_{r_i})$ belong to D_{r_i}. ∎

From the definition of π^{-1}, the elements belonging to D_{r_i} thus occur in a contiguous portion of the array π^{-1} between locations r_{i-1} and r_i.

In the set D_{r_i}, we now consider points c_{ij} such that each c_{ij} is dominated by only r_i and by no other integer in D_{r_i}. Equivalently, if we consider region D_{r_i} and remove r_i from the graph, the elements c_{ij} are the roots. In this context(i.e considering c_{ij}'s to be the roots), the regions $D_{c_{ij}}$ may be similarly defined i.e, the region $D_{c_{ij}}$ consists of all elements which belong to D_{r_i} and have c_{ij} as their closest dominating root in D_{r_i}.

Lemma 5.8 The set $D_{c_{ij}}$ consists exactly of all the integers in the range $[y_{c_{i(j-1)}}..y_{c_{ij}}]$.

Proof: The proof is similar to that of Lemma 5.7. ▮

Thus, the elements belonging to each $D_{c_{ij}}$ also occur in contiguous locations (within the region defined by their corresponding D_{r_i}) in π^{-1}.

In the set $D_{c_{ij}}$ we define $lx[ij]$ to be the integer with largest x-coordinate(i.e position), and $sx[ij]$ to be the integer with second largest x-coordinate.

Lemma 5.9 The integers $lx[ij]$ and $sx[ij]$ can be computed for all $D_{c_{ij}}$ in $\tau(n)$ time with $\rho(n)$ processors.

Proof: From Lemma 5.8, π^{-1} is partitioned such that $D_{c_{ij}}$ is exactly the set of integers between $c_{i(j-1)}$ and c_{ij}. So, $lx[ij]$ is simply the maximum value in the subarray $\pi^{-1}[(c_{i(j-1)})..(c_{ij})]$ and can be computed using prefix-maxima routine. ▮

In the set D_{r_i} (for some i), corresponding to c_{ij} we define the set $C_l(ij)$ as follows: $C_l(ij) = \{c_{ik} \mid k < j \text{ and } x[c_{ij}] < lx[ik]\}$. Intuitively, $C_l(ij)$ consists of c_{ik}'s for which $D_{c_{ik}}$ contains an element ($lx[ik]$) dominated by c_{ij}; i.e., there is an edge between $lx[ik]$ and c_{ij} in G. Thus, by a result of Liang et.al. [22], the elements of $D_{c_{ik}}$ together with elements of $D_{c_{ij}}$ will form a biconnected component. Term $lspan[c_{ij}]$ will denote the y-coordinate (i.e value) of the minimum element in $C_l(ij)$. Thus, if $k' = \min\{k \mid c_{ik} \in C_l(ij)\}$, then $lspan[c_{ij}] = y[c_{ik'}]$,

Lemma 5.10 For each c_{ij}, $lspan[c_{ij}]$ may be computed in $\tau(n)$ time with $\rho(n)$ processors.

Proof: For some i we consider the computation of all $lspan[c_{ij}]$'s. Recall, the elements c_{ij} are increasing both in value and in position. Since we need to compute the first c_{ik} such that $lx[ik] > x[c_{ij}](\pi^{-1}(c_{ij}))$, an obvious way is to compute the prefix-maxima over the values (x-coordinate) of $lx[ik]$ in D_{r_i} and find out the first location where the prefix-maxima value is more than $x[c_{ij}]$ by a binary search on the prefix-maxima values. However, we can improve on the resource bounds by noting that after computing prefix-maxima we have got a sorted(non-decreasing) array of integers in the range [1..n]. So, to locate the positions of all c_{ij} in this array, we can merge the two arrays and marking each of the prefix-maximum values as 1 and the c_{ij} elements as 0, we just need to compute for each 0 the nearest 1 to the right. This will give the identity of the desired c_{ik}. Now, the regions D_{r_i} being disjoint, the computation for all of them can be carried out simultaneously. The total sum of the number of elements in all regions D_{r_i} is $O(n)$ only, and so the computation can be carried out within the resource bounds of prefix-maxima and merge routines. ∎

An interval graph [14] consists of a set of intervals $I_i = [a_i, b_i]$, ($a_i < b_i$) on the real line: there is a vertex in the graph for each interval and there is an edg[illegible]ween two vertices if their corresponding intervals have a non-empty intersection. We consider the set D_{r_i} and construct an interval graph G_i in the following manner: corresponding to each c_{ij} belonging to the region D_{r_i}, there is an interval $[lspan[c_{ij}], y[c_{ij}]]$ in G_i. From [22] the connected components in G_i are biconnected components in the region D_{r_i}. Note that in the set D_{r_i}, the right end-points of the intervals occur in sorted order and the left end-points are also sorted. Computing the connected components in an interval graph is equivalent to computing prefix-minima after sorting the end-points. In this case, sorting of end-points is equivalent to merging of two sorted arrays of integers. Thus, the routines prefix-minima and merge together solve the connected component problem on each G_i. As, the elements belonging to the D_{r_i}'s (and hence G_i's) are disjoint and the total number of intervals is $O(n)$ only, this computation can also be carried out in $\tau(n)$ time with $\rho(n)$ processors.

Once the connected components in G_i have been computed, we retain only those c_{ij}'s which are the minimum elements in some connected component and remove the other c_{ik}'s along with the elements in regions $D_{c_{ik}}$.

In this reduced graph, corresponding to c_{ij} we define the set $C_d(ij)$ as follows: $C_d(ij) = \{c_{kl} \mid k < i \text{ and } x[c_{ij}] < sx[kl]\}$. Intuitively, $C_d(ij)$ consists of c_{kl}'s for which $D_{c_{ik}}$ contains two elements ($lx[ik]$ and $sx[kl]$) dominated by c_{ij}; i.e., there is an edge between $lx[ik]$ and c_{ij} and also between $sx[ik]$ and c_{ij} in G. Thus, by a result of Liang et.al. [22], the elements of $D_{c_{kl}}$ together with elements of $D_{c_{ij}}$ will form a biconnected component. Term $dspan[c_{ij}]$ will denote the y-coordinate(i.e value) of the minimum element in $C_d(ij)$. Thus, if $k' = \min\{k \mid c_{kl} \in C_l(ij)\}$ and $l' = \min\{l \mid c_{k'l} \in C_l(ij)\}$ $dspan[c_{ij}] = y[c_{k'l'}]$.

Lemma 5.11 For all c_{ij}'s, $dspan[c_{ij}]$ may be computed in $\tau(n)$ time with $\rho(n)$ processors.

Proof: Consider computation of $dspan[c_{ij}]$ for an element c_{ij}: in the reduced π^{-1}, if we consider only sx's, the computation effectively locates the least sx for which the position in π is greater than position of c_{ij} in π. Thus, in reduced π if we compute suffix-minima on only sx's, the suffix-minima value at position $\pi^{-1}(c_{ij})$ indicates this desired element; hence we can solve the problem within the bounds of prefix-maxima routine. ∎

From [22], the connected components in G' are biconnected components in the given permutation graph.

Lemma 5.12 The connected components of G' can be computed in $\tau(n)$ time with $\rho(n)$ processors.

Proof: In G_i, both the end-points were sorted, but in G' only the right end-points are sorted. However, in an interval graph, if any one of the end-points are sorted, the connected components may be computed using prefix-maxima [28]. ∎

Thus we have the following theorem.

Theorem 5.2 The biconnected components of a permutation graph can be computed in $O(\log n)$ time using $\frac{n}{\log n}$ processors on the CREW PRAM or $O(\log\log\log n)$ time using $\frac{n}{\log\log\log n}$ processors on the COMMON(TOLERANT) CRCW PRAM.

Proof: From the previous discussion, the costliest steps needed in the algorithm are prefix-maxima and merge which give rise to the stated resource bounds. ∎

Chapter 6

Concluding Remarks

In Chapter 2, constant time parallel algorithms have been obtained for some problems in interval graphs on the reconfigurable mesh model. However, there are some problems in interval graphs which can not be solved in constant time, for e.g. finding the shortest path between a pair of vertices, even with $O(n^2)$ processors. List-ranking within the stated resource bounds appears to be a bottleneck. .

For the most vital edge problem considered in Chapter 3, the time taken by the sequential algorithm is $O(\min(m + n \log n, m\alpha(m, n))$ while the time-processor product of the parallel algorithms is $O(\min(n^2, m \log m))$. Hence there is still some scope for further improvement.

In Chapter 4, the BFS algorithm runs optimally in $O(\log n)$ time on the EREW PRAM which is the weakest PRAM model- on this model, as $\log n$ is a lower bound for most problems, there appears to be little scope for improvement. However, on some other model, a faster parallel algorithm may be possible. For the DFS algorithm, as repeatedly mentioned, the bottleneck is computation of longest increasing subsequence. Obtaining an optimal parallel algorithm matching the lower bound of $O(n \log \log n)$ is an open problem.

In Chapter 5, the connected and biconnected components for a permutation graph are computed optimally. The algorithm for biconnected components could possibly be extended to optimally compute the k-connected components of a permutation graph.

Bibliography

[1] K. Abrahamson, N. Dadoun, D.G. Kirkpatrick, and T. Przytycka. A simple parallel tree contraction algorithm. *Journal of Algorithms*, 10(2):287–302, 1989.

[2] A. Aggarwal, B. Chazelle, L. Guibas, C. O'Dunlaing, and C. Yap. Parallel computational geometry. *Algorithmica*, 3:293–328, 1988.

[3] S.G Akl. *The design and analysis of parallel algorithms.* Prentice Hall, Englewood Cliffs, 1989.

[4] R.J Anderson and G.L. Miller. Deterministic parallel list ranking. *Algorithmica*, 6(6):859–868, 1991.

[5] M.J. Atallah, R. Cole, and M. Goodrich. Cascading divide-and-conquer: a technique for designing parallel algorithms. *SIAM. Jl. of Comput.*, 18(3):499–532, 1989.

[6] C Berge. Perfect graphs. In D.R. Fulkerson, editor, *Studies in graph theory Part I*, pages 1-22, 1975.

[7] O. Berkman, J. Ja'Ja', S. Krishnamurthy, R. Thurimella, and U Vishkin. Some triply logarithmic parallel algorithms. In *Proceedings IEEE FOCS*, pages 871-881. IEEE, 1990.

[8] O. Berkman, B. Schieber, and U Vishkin. Optimal doubly logarithmic parallel algorithms for finding all nearest smaller values. *Journal of Algorithms*, 14(3):344–370, 1993.

[9] O. Berkman and U Vishkin. On parallel integer merging. *Information and Computation*, 106(2):266–285, Oct 1993.

[10] B. Chazelle and L. Guibas. Fractional cascading i: a data structuring technique. *Algorithmica*, 1:133–162, 1986.

[11] R. Cole and U. Vishkin. Approximate and exact parallel scheduling with application to list, tree and graph problems. In *FOCS*, pages 478–491. IEEE, 1986.

[12] N.S Deo. *Graph theory with applications to Engineering and Computer Science.* Series in Automatic Computation. Prentice Hall, 1974.

[13] S. Even, A. Pnueli, and A. Lempel. Permutation graphs and transitive graphs. *Journal of ACM*, 19(3):400–410, 1972.

[14] M.C. Golumbic. *Algorithmic Graph theory and perfect graphs.* New York: Academic, 1980.

[15] L.H. Hsu, R.H. Jan, Y.C. Lee, C.N. Hung, and M.S. Chern. Finding the most vital edge with respect to minimum spanning tree in weighted graphs. *IPL*, 39(5):277 - 281, 1991.

[16] O.H. Ibarra and Q. Zheng. An optimal shortest path parallel algorithm for permutation graphs. *Jl.of Parallel and Dist. Computing*, 24:94–99, 1995.

[17] K. Iwama and Y. Kambayushi. A simpler parallel algorithm for graph connectivity. *Journal of Algorithms*, 16:190–217, 1994.

[18] Iwano.K. and Katoh.N. Efficient algorithms for finding the most vital edge of a minimum spanning tree. *IPL*, 48(3):211 - 213, 1993.

[19] J. Jang and V.K. Prasanna. An optimal sorting algorithm on reconfigurable mesh. In *Proc.6'th Intl. Parallel Processing Symp.*, pages 130- 137. IEEE, 1992.

[20] D.B. Johnson and P. Metaxas. A parallel algorithm for computing minimum spanning tree. In *SPAA*, pages 363–372, 1992.

[21] H. Li and M. Aresca. Polymorphic-torus architecture for computer vision. *IEEE Trans. Pattern Anal. Mach. Intell.*, 1(3):233- 243, Mar 1989.

[22] Y.D. Liang, S.K. Dhall, and S. Lakshmivarahan. Efficient parallel algorithms for finding biconnected components of some intersection graphs. In *ACM Computer Science Conference*, pages 48–52. ACM, 1991.

[23] Lu Mi. Parallel computation of longest-common-subsequence. *LNCS*, 468(1):385–394, 1990.

[24] R. Miller, V.K. Prasanna, D.I. Reisis, and Q.F. Stout. Image computations on reconfigurable VLSI arrays. In *Proc. IEEE Comput. Soc. Conf. Comput. Vision*

Pattern Recog, pages 926- 930. IEEE, 1988.

[25] R. Miller, V.K. Prasanna, D.I. Reisis, and Q.F. Stout. Meshes with reconfigura buses. In *Proc. 5th MIT Conf. on Advanced Research in VLSI*, pages 163- 1 1988.

[26] D. Nath, S.N. Maheshwari, and P.C.P. Bhatt. Efficient VLSI networks for paral processing based on orthogonal trees. *IEEE Trans. on Comput.*, 32(6):569 – 57 June 1983.

[27] S. Olariu, J.L. Schwing, and J. Zhang. Optimal parallel algorithms for prol lems modelled by a family of intervals. *IEEE Trans. on Parallel and Dist. Sys* 3(3):364- 374, May 1992.

[28] N.M Rao. Parallel algorithms for some problems on perfect graphs. Master's thesis, I.I.T.Kanpur, 1994.

[29] C. Rhee, Y.D. Liang, S.K. Dhall, and S. Lakshmivarahan. Efficient algorithms for finding depth-first and breadth-first search trees in permutation graphs. *IPL*, 49(1):45–50, 1994.

[30] S. Saxena. Two-coloring linked lists is NC^1 - complete for log space. *IPL*, 49(2):73 – 79, 1994.

[31] R.E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM. Jl. of Comput.*, 14(4):862 - 874, Nov 1984.

[32] B.F. Wang and G.H. Chen. Constant time algorithms for the transitive closure and some related graph problems on processor arrays with reconfigurable bus systems. *IEEE Trans. on Parallel and Dist. Sys.*, 1(4):500- 507, Oct 1990.

[33] B.F. Wang and G.H. Chen. Two-dimensional processor array with reconfigurable bus system is at least as powerful as CRCW model. *IPL*, 36(1):31 – 36, 1990.